Branch and Peg Algorithms for the Simple Plant Location Problem

Boris Goldengorin
Diptesh Ghosh*
Gerard Sierksma
Faculty of Economic Sciences, University of Groningen, The Netherlands.

SOM-theme A Primary Processes within Firms

Abstract

The simple plant location problem is a well-studied problem in combinatorial optimization. It is one of deciding where to locate a set of plants so that a set of clients can be supplied by them at the minimum cost. This problem often appears as a subproblem in other combinatorial problems. Several branch and bound techniques have been developed to solve these problems. In this paper we present a few techniques that enhance the performance of branch and bound algorithms. The new algorithms thus obtained are called branch and peg algorithms, where pegging refers to assigning values to variables outside the branching process. We present exhaustive computational experiments which show that the new algorithms generate less than 60% of the number of subproblems generated by branch and bound algorithms, and in certain cases require less than 10% of the execution times required by branch and bound algorithms.

Keywords: Simple Plant Location Problem, Pseudo-Boolean representation, Beresnev function, Branch and Bound, Preprocessing, Pegging.

---

* Corresponding Author: Faculty of Economic Sciences, University of Groningen, P.O. Box 800, 9700AV Groningen, The Netherlands. Email: D.Ghosh@eco.rug.nl

1. Introduction

Given sets $I = \{1, 2, \ldots, m\}$ of sites in which plants can be located, $J = \{1, 2, \ldots, n\}$ of clients, a vector $F = (f_i)$ of fixed costs for setting up plants at sites $i \in I$, a matrix $C = [c_{ij}]$ of transportation costs from $i \in I$ to $j \in J$, and an unit demand at each client site, the Simple Plant Location Problem (SPLP) is the problem of finding a set S, $\emptyset \subset S \subseteq I$, at which plants can be located so that the total cost of satisfying all client demands is minimal. The costs involved in meeting the client demands include the fixed costs of setting up plants, and the transportation cost of supplying clients from the plants that are set up. A detailed introduction to this problem appears in Cornuejols *et al.* [12]. The SPLP forms the underlying model in several combinatorial problems, like set covering, set partitioning, information retrieval, simplification of logical Boolean expressions, airline crew scheduling, vehicle despatching (Christofides [6]), assortment (Beresnev *et al.* [5], Goldengorin [17], Jones *et al.* [23], Pentico [29, 30], Tripathy *et al.* [33]), and is a subproblem for various location analysis problems (Revelle and Laporte [31]).

The SPLP is NP-hard (Cornuejols *et al.* [12]), and many exact and heuristic algorithms to solve the problem have been discussed in the literature. Most of the exact algorithms are based on a mathematical programming formulation of the SPLP. Direct approaches (Schrage [32], Morris [27]) use a branch and bound approach and the strong linear programming relaxation (SLPR) for computing bounds. However such approaches cannot always output an optimal solution to average-sized SPLP instances in reasonable time. More efficient solution approaches to the SPLP are based on Langrangian duality (Held *et al.* [22], Beresnev *et al.* [5]). Computational experience of solving the Lagrangian dual using subgradient optimization have been reported in Cornuejols *et al.* [10] and Cornuejols and Thizy [11], and using Dantzig-Wolfe decomposition in Garfinkel *et al.* [15]. Computer codes for solving medium sized SPLP using mixed-integer programming systems are also available (Martin and Schrage [26], Van Roy and Wolsey [35]). Polyhedral results for the SPLP polytope have been reported in Trubin [34], Balas and Padberg [1], Mukendi [28], Cornuejols *et al.* [9], Krarup and Pruzan [24], Cho *et al.* [7], and Cho *et al.* [8]. In theory, these results allow us to solve the SPLP by applying the simplex algorithm to SLPR, with the additional stipulation that a pivot to a new extreme point is allowed only when this new extreme point is integral. Although some computational experience using this method has been reported in the literature (Guignard and Spielberg [20]), efficient implementations of this pivot rule are not available. Results of computational experiments with Lagrangian heuristics for medium sized instances of the SPLP have been reported in Beasley [2]. Large-sized SPLP instances

can be solved using algorithms based on refinements to a dual-ascent heuristic procedure to solve the dual of LP-relaxation of the SPLP (Körkel [25]), combined with the use of the complementary slackness conditions to construct primal solutions (Erlenkotter [14]). Preprocessing rules, which form a common component in branch and bound implementations, are strangely never explicitly discussed in the available literature on computational experiences with the SPLP.

The pseudo-Boolean representation of the SPLP facilitates the construction of rules to reduce the size of SPLP instances (Beresnev *et al.* [5], Cornuejols *et al.* [12], Dearing *et al.* [13], Goldengorin *et al.* [19], Veselovsky [36], etc.). These rules have never been used in the explicit description of any algorithm for the SPLP in the available literature. In this paper we propose branch and bound based algorithms called branch and peg algorithms, which use these rules, not only for preprocessing, but also as a tool either to solve a subproblem or to reduce its size. They also use information from a pseudo-Boolean representation of the SPLP to compute efficient branching functions. For the sake of simplicity, we use a common depth first branch and bound scheme in our implementations and a simple combinatorial bound, but the concepts developed herein can easily be implemented in any of the algorithms cited above. Our implementations satisfy our purpose in this paper, which is to test the superiority of branch and peg algorithms over branch and bound algorithms using the same bound.

The remainder of this paper is organized as follows. In Section 2 we describe a pseudo-Boolean representation of the SPLP. In Section 3 we describe branch and peg algorithms in detail, and our computational experience with them on randomly generated instances as well as some benchmark instances from Beasley [2] as benchmark instances for our computational experiments. We summarize the paper in Section 4 and propose directions for further research.

2. A Pseudo-Boolean Approach to SPLP

The pseudo-Boolean approach to solving the SPLP (Hammer [21], Beresnev [4]) is a penalty-based approach that relies on the fact that any instance of the SPLP has an optimal solution in which each client is supplied by exactly one plant. This implies, that in an optimal solution, each client will be served fully by the plant closest to it. Therefore, it is sufficient to determine the sites where plants are to be located, and then use a minimum weight matching algorithm to assign clients to plants.

An instance of the SPLP can be described by a m-vector $F = (f_i)$, and a $m \times n$ matrix $C = [c_{ij}]$; $m, n \geq 1$. We assume that elements of F and C are nonnegative. We will use the $m \times (n+1)$ *augmented matrix* $[F|C]$ as a shorthand for describing an instance of the SPLP. The total cost $f_{[F|C]}(S)$ associated with a solution S consists of two components, namely the fixed costs $\sum_{i \in S} f_i$ and the transportation costs $\sum_{j \in J} \min\{c_{i,j} | i \in S\}$; i.e.

$$f_{[F|C]}(S) = \sum_{i \in S} f_i + \sum_{j \in J} \min\{c_{i,j} | i \in S\},$$

and the SPLP is the problem of finding

$$S^\star \in \arg\min\{f_{[F|C]}(S) : \emptyset \subset S \subseteq I\}. \tag{1}$$

In the remainder of this section we describe the pseudo-Boolean formulation of the SPLP due to Beresnev [4].

A $m \times n$ *ordering matrix* $\Pi = [\pi_{ij}]$ is a matrix each of whose columns $\Pi_j = (\pi_{1j}, \ldots, \pi_{mj})^T$ define a permutation of $1, \ldots, m$. Given a transportation matrix C, the set of all ordering matrices Π such that $c_{\pi_{1j}j} \leq c_{\pi_{2j}j} \leq \cdots \leq c_{\pi_{mj}j}$, for $j = 1, \ldots, n$, is denoted by $perm(C)$.

Defining

$$y_i = \begin{cases} 0 & \text{if } i \in S \\ 1 & \text{otherwise,} \end{cases} \quad \text{for each } i = 1, \ldots, m \tag{2}$$

we can indicate any solution S by a vector $\mathbf{y} = (y_1, y_2, \ldots, y_m)$. The fixed cost component of the total cost can be written as

$$\mathcal{F}_F(\mathbf{y}) = \sum_{i=1}^{m} f_i(1 - y_i). \tag{3}$$

Given a transportation cost matrix C, and an ordering matrix $\Pi \in perm(C)$, we can denote differences between the transportation costs for each $j \in J$ as

$$\begin{aligned} \Delta c[0, j] &= c_{\pi_{1j}j}, \quad \text{and} \\ \Delta c[l, j] &= c_{\pi_{(l+1)j}j} - c_{\pi_{lj}j}, \quad l = 1, \ldots, m-1. \end{aligned}$$

Then, for each $j \in J$,

$$\begin{aligned} \min\{c_{i,j} | i \in S\} &= \Delta c[0, j] + \Delta c[1, j] \cdot y_{\pi_{1j}} + \Delta c[2, j] \cdot y_{\pi_{1j}} \cdot y_{\pi_{2j}} \\ &\quad + \cdots + \Delta c[m-1, j] \cdot y_{\pi_{1j}} \cdots y_{\pi_{(m-1)j}} \\ &= \Delta c[0, j] + \sum_{k=1}^{m-1} \Delta c[k, j] \cdot \prod_{r=1}^{k} y_{\pi_{rj}}, \end{aligned}$$

so that the transportation cost component of the cost of a solution $\mathbf{y}$ corresponding to an ordering matrix $\Pi \in perm(C)$ is

$$\mathcal{T}_{C,\Pi}(\mathbf{y}) = \sum_{j=1}^{n} \left\{ \Delta c[0, j] + \sum_{k=1}^{m-1} \Delta c[k, j] \cdot \prod_{r=1}^{k} y_{\pi_{rj}} \right\}. \tag{4}$$

Combining (3) and (4), the total cost of a solution $\mathbf{y}$ to the instance $[F|C]$ corresponding to an ordering matrix $\Pi \in perm(C)$ is given by the pseudo-Boolean polynomial

$$\begin{aligned} f_{[F|C],\Pi}(\mathbf{y}) &= \mathcal{F}_F(\mathbf{y}) + \mathcal{T}_{C,\Pi}(\mathbf{y}) \\ &= \sum_{i=1}^{m} f_i(1 - y_i) + \sum_{j=1}^{n} \left\{ \Delta c[0, j] + \sum_{k=1}^{m-1} \Delta c[k, j] \cdot \prod_{r=1}^{k} y_{\pi_{rj}} \right\}. \end{aligned} \tag{5}$$

It can be shown (Goldengorin *et al.* [19]) that the total cost function $f_{[F|C],\Pi}(\cdot)$ is identical for all $\Pi \in perm(C)$. We call this pseudo-Boolean polynomial the *Beresnev function* $\mathcal{B}_{[F|C]}(\mathbf{y})$ corresponding to the SPLP instance $[F|C]$ and $\Pi \in perm(C)$. In other words

$$\mathcal{B}_{[F|C]}(\mathbf{y}) = f_{[F|C],\Pi}(\mathbf{y}) \text{ where } \Pi \in perm(C). \tag{6}$$

We can formulate (1) in terms of Beresnev functions as

$$\mathbf{y}^{\star} \in \arg\min\{\mathcal{B}_{[F|C]}(\mathbf{y}) : \mathbf{y} \in \{0, 1\}^m, \mathbf{y} \neq \mathbf{1}\}. \tag{7}$$

Beresnev functions assume a key role in the development of the algorithms described in the next section.

3. Branch and Peg Algorithms

In this section we describe enhancements to branch and bound (BnB) algorithms to solve SPLP instances. The enhanced algorithms use a rule based on Beresnev functions to *peg* variables in subproblems, i.e. to determine before branching, whether plants will (or will not) be located at certain sites in the current subproblem. This rule, used on the initial instance, forms a preprocessing rule. The other enhancement that we propose, is the use of Beresnev functions to devise effective branching functions. The new algorithms are collectively called *branch and peg* (BnP) algorithms. We use a depth first search strategy in our algorithms, but the concepts can be used unchanged in best first search.

The following terms are used in the remainder of this section. A *solution* to a SPLP instance with $|I| = m$ and $|J| = n$ is a vector $\mathbf{y}$ of length m, defined on the alphabet $\{0, 1, \#\}$. $y_j = 0$ (respectively, $y_j = 1$) indicates that a plant is located (respectively, not located) at the site with index j. $y_j = \#$ indicates that no decision regarding locating a plant at the site with index j has been incorporated in the solution. A solution $\mathbf{y}$ with $y_j = \#$ for some j is called a *partial solution*, while all other solutions are called *complete solutions*. The process of setting the value of y_j for any index j in any partial solution $\mathbf{y}$ to 0 or 1 is called *pegging the variable*. If $y_j = \#$, then y_j is called an *open* variable, and if $y_j = 0$ or 1, it is called a *pegged* variable.

The pseudo-Boolean representation of the SPLP allows us to develop rules using which we can peg certain variables in a solution by examining the coefficients of the Beresnev functions. The rule that we use here is described in Goldengorin *et al.* [19] as a preprocessing rule. We assume, without loss of generality, that the instance is not separable, i.e. we cannot partition I into sets I_1 and I_2, and J into sets J_1 and J_2, such that the transportation costs from sites in I_1 to clients in J_2, and from sites in I_2 to clients in J_1 are not finite. We also assume without loss of generality, that the site indices are arranged in non-increasing order of $f_i + \sum_{j \in J} c_{ij}$ values. We include the proof of correctness of this rule for the sake of completeness.

Pegging Rule (Goldengorin *et al.* [19]) *Let $\mathcal{B}_{[F|C]}(\mathbf{y})$ be the Beresnev function corresponding to the SPLP instance $[F|C]$ in which like terms have been aggregated. Let $a_k = (\sum_{j:\pi_{1j}=k} \Delta c[1, j]) - f_k$ be the coefficient of the linear term corresponding to y_k and let $t_k = \sum^{n}_{\substack{j=1 \\ j:k \in \{\pi_{1j},\ldots,\pi_{pj}\}}} \sum_{p=2}^{m-1} \Delta c[p, j]$ be the sum of the coefficients of all non-linear terms containing y_k for each site index k. Then the following hold.*

- *(a) If $a_k \geq 0$, then there is an optimal solution $\mathbf{y}^\star$ in which $y^\star_k = 0$, else*
- *(b) if $a_k + t_k \leq 0$, then there is an optimal solution $\mathbf{y}^\star$ in which $y^\star_k = 1$, provided $y^\star_i \neq 1$ for some $i \neq k$.*

PROOF.

(a) Suppose $a_k \geq 0$. Let us consider a solution $\mathbf{y}$ in which $y_k = 1$ and a solution $\mathbf{y}'$ in which $y'_i = y_i$ for each $i \neq k$, and $y'_k = 0$. Now $\mathcal{B}_{[F|C]}(\mathbf{y}) - \mathcal{B}_{[F|C]}(\mathbf{y}') \geq a_k \geq 0$. Hence $\mathbf{y}'$ is preferable to $\mathbf{y}$. This shows that $y_k = 0$ in an optimal solution.

(b) Next suppose that $a_k + t_k \leq 0$. Consider two solutions $\mathbf{y}$ and $\mathbf{y}'$, such that $y_i = y'_i$ for each $i \neq k$, $y_k = 0$, and $y'_k = 1$. Then

$$\begin{aligned}
&\mathcal{B}_{[F|C]}(\mathbf{y}') - \mathcal{B}_{[F|C]}(\mathbf{y}) \\
&= \{\sum_{i=1}^{m} f_i(1 - y'_i) + \sum_{j=1}^{n}\sum_{p=1}^{m-1} \Delta[p, j] \prod_{r=1}^{p} y'_{\pi_{rj}}\} - \\
&\quad \{\sum_{i=1}^{m} f_i(1 - y_i) + \sum_{j=1}^{n}\sum_{p=1}^{m-1} \Delta[p, j] \prod_{r=1}^{p} y_{\pi_{rj}}\} \\
&= \{-f_k y'_k - \underbrace{\sum_{\substack{i=1 \\ i\neq k}}^{m} f_i y'_i}_{A} + \sum_{\substack{j=1 \\ j:k\in\{\pi_{1j},\ldots,\pi_{pj}\}}}^{n} \sum_{p=1}^{m-1} \Delta[p, j] \prod_{r=1}^{p} y'_{\pi_{rj}} + \\
&\quad \underbrace{\sum_{\substack{j=1 \\ j:k\notin\{\pi_{1j},\ldots,\pi_{pj}\}}}^{n} \sum_{p=1}^{m-1} \Delta[p, j] \prod_{r=1}^{p} y'_{\pi_{rj}}}_{B}\} - \\
&\quad \{-f_k y_k - \underbrace{\sum_{\substack{i=1 \\ i\neq k}}^{m} f_i y_i}_{C} + \sum_{\substack{j=1 \\ j:k\in\{\pi_{1j},\ldots,\pi_{pj}\}}}^{n} \sum_{p=1}^{m-1} \Delta[p, j] \prod_{r=1}^{p} y'_{\pi_{rj}} + \\
&\quad \underbrace{\sum_{\substack{j=1 \\ j:k\notin\{\pi_{1j},\ldots,\pi_{pj}\}}}^{n} \sum_{p=1}^{m-1} \Delta[p, j] \prod_{r=1}^{p} y_{\pi_{rj}}}_{D}\} \qquad (8)
\end{aligned}$$

Notice that the terms marked A and C cancel each other since $y_i = y'_i$ when $i \neq k$, as do the terms marked B and D. Cancelling these terms and setting $y_k = 0$ and $y'_k = 1$ in (8) we obtain

$$\begin{aligned}
&\mathcal{B}_{[F|C]}(\mathbf{y}') - \mathcal{B}_{[F|C]}(\mathbf{y}) \\
&= \{-f_k + \sum_{\substack{j=1 \\ j:k\in\{\pi_{1j},\ldots,\pi_{pj}\}}}^{n} \sum_{p=1}^{m-1} \Delta c[p, j] \prod_{r=1}^{p} y'_{\pi_{rj}}\} \qquad (9)
\end{aligned}$$

which, on separating the linear and non-linear terms

$$= \{a_k + \sum_{\substack{j=1 \\ j:k\in\{\pi_{1j},\ldots,\pi_{pj}\}}}^{n} \sum_{p=2}^{m-1} \Delta c[p, j] \prod_{r=1}^{p} y'_{\pi_{rj}}\}. \tag{10}$$

An upper bound to (10) is $a_k + t_k$, which is obtained by setting $y'_i = 1$ for each $i \in I$, since all non-linear terms in the Beresnev function have non-negative coefficients. Thus

$$\mathcal{B}_{[F|C]}(\mathbf{y}') - \mathcal{B}_{[F|C]}(\mathbf{y}) \leq (a_k + t_k) \leq 0. \tag{11}$$

Hence $\mathbf{y}'$ is preferable to $\mathbf{y}$. This shows that $y_k = 1$ in an optimal solution. Of course, if $y_i^\star = 1$ for all $i \neq k$, then setting $y_k^\star$ to 1 would yield an infeasible solution.

■

Note that $t_k \geq 0$ for each index k, since the non-linear terms of the Beresnev function are non-negative. Thus $a_k + t_k \leq 0 \Rightarrow a_k \leq 0$. If $t_k = 0$, then there is a possibility of a_k being equal to zero, but this possibility is taken care of in the first part of the rule.

The importance of the ordering of the site indices is demonstrated in the following lemma.

Lemma 3.1 *If $a_k < 0$ and $a_k + t_k \leq 0$ for each $k \in I$ in $\mathcal{B}_{[F|C]}(\mathbf{y})$ for the SPLP instance $[F|C]$, then an optimal solution would be $(1, 1, \ldots, 1, 0)$ assuming that the site indices are arranged in non-increasing order of $f_i + \sum_{j\in J} c_{ij}$ values.*

PROOF. Let us initially relax the constraint $\mathbf{y} \neq \mathbf{1}$ in (7). In such a case it is easy to see that the optimal solution would be $\mathbf{y} = \mathbf{1}$ (from (11)). If we reimpose the constraint, we need to set one or more y_k values to 0. Changing $y_k = 0$ for any variable $k \in I$ increases the value of the Beresnev function by $f_k + \sum_{j\in J} c_{kj}$. Note that setting $y_k = 0$ does not affect the non-positive nature of $a_i + t_i$, $i \neq k$, since this operation does not affect a_i and can only reduce the value of t_i. Also note that setting any additional variable y_i, $i \neq k$ to 0 cannot reduce the value of Beresnev function since $a_i < 0$ and $a_i + t_i \leq 0$ for each $i \neq k$. The result follows. ■

The lemma above is illustrated by the following example. Consider a SPLP instance $[F|C]$, $m = n = 3$, in which $F = (99, 100, 98)$ and

$$C = \begin{bmatrix} 0 & 10 & 13 \\ 10 & 0 & 16 \\ 13 & 16 & 0 \end{bmatrix}$$

The Beresnev function for this instance is

$$\mathcal{B}_{[F|C]} = 297 - 89y_1 - 90y_2 - 85y_3 + 9y_1y_2 + 3y_1y_3.$$

It is clear that $a_k < 0$ and $a_k + t_k < 0$ for $k = 1$, 2, and 3. Therefore the Pegging Rule will solve this instance completely, set $y_k = 1$ the first two sites it encounters, and set $y_k = 0$ for the last site. However, the solution would be correct only if the last site encountered has the lowest $f_i + \sum_{j \in J} c_{ij}$ value, i.e., if site 1 is considered after sites 2 and 3. In general therefore, the sites i should be ordered in non increasing values of $f_i + \sum_{j \in J} c_{ij}$.

In the remainder of this section, we will assume the existence of a procedure $PegPartialSolution$, which implements the Pegging Rule. It accepts a partial solution as input, repeatedly applies the Pegging Rule above until no further pegging is possible, and returns the solution thus obtained. (Note that the solution returned by this procedure will, in general, be a partial solution.) $PegPartialSolution$ will be used for preprocessing in both BnB and BnP algorithms, as well as for pegging variables in partial solutions in the BnP search tree. We will call the solution obtained after preprocessing, (implemented by running $PegPartialSolution$ on the partial solution $(\#\#\cdots\#)$,) the *initial solution.* This solution forms the root of the BnP and BnB search trees.

The choice of the variable to branch on is critical for the success of a branch and bound scheme. The following trivial branching function can be used in the absence of any prior knowledge regarding the suitability of the variable to branch on.

Branching Function 1 *Return the open variable with the minimum index in the current subproblem.*

However, we could use information from the coefficients of the Beresnev function to create more effective branching functions.

Consider a subproblem $\mathcal{P}$ in which the partial solution, after being pegged by the $PegPartialSolution$ procedure, is $\mathbf{y}$. For each variable y_k in $\mathbf{y}$ that is still open, let $\mathbf{y}^{k0}$

be obtained by forcibly pegging y_k to 0, and running $PegPartialSolution$ on the resultant solution; let ϕ_{k0} be the number of open variables in $\mathbf{y}^{k0}$. Similarly, let $\mathbf{y}^{k1}$ be obtained by forcibly pegging y_k to 1 in $\mathbf{y}$, and running $PegPartialSolution$ on the resultant solution; let ϕ_{k1} be the number of open variables in $\mathbf{y}^{k1}$. If we want to obtain a quick upper bound for the objective value of the optimal solution to $\mathcal{P}$ by solving its subproblems by pegging, then $\phi_k = \min(\phi_{k0}, \phi_{k1})$ is a good measure of the suitability of y_k as a branching variable. (Other combinations of ϕ_{k0} and ϕ_{k1}, such as $\frac{\phi_{k0}+\phi_{k1}}{2}$ could also be used, but our preliminary experimentation shows that these do not cause significant differences in the results obtained.) A branching function based on such a measure, can be expected to generate relatively few subproblems while solving a SPLP instance. However, the calculations involved would, in general, take excessive time. As a compromise therefore, we could use a branching function that generates the ordering of the indices once for the initial solution and uses it for all subproblems. This branching function is described below.

Branching Function 2 *For each open variable y_k in the initial solution $\mathbf{y}$, let ϕ_{k0} (respectively, ϕ_{k1}) be the number of open variables in the solution obtained by setting $y_k = 0$ (respectively, $y_k = 1$) in the initial solution and running $PegPartialSolution$ on it. Define the fitness ϕ_j of the variable y_j as*

$$\phi_j = \begin{cases} \min(\phi_{j0}, \phi_{j1}) & \text{if } y_j \text{ is open in the initial solution,} \\ \infty & \text{otherwise.} \end{cases}$$

Return the open variable y_j for which ϕ_j is minimal.

A third branching function may be devised in the following manner. Consider a subproblem $\mathcal{P}$ in which the partial solution, after being pegged by $PegPartialSolution$, is $\mathbf{y}$. Recall that the Pegging Rule pegs any variable y_k with $a_k \geq 0$ to 0, and any variable y_k with $t_k + a_k \leq 0$ to 1. Therefore, for each variable y_k that has not been pegged we conclude that $a_k < 0$ and $t_k + a_k > 0$. y_k would have been pegged to 0 in this solution if the coefficient of linear term involving y_k in the Beresnev function would have been increased by $-a_k$. It would have been pegged to 1, if the same coefficient would have been decreased by $t_k + a_k$. Therefore we could use $\phi_k = \max(-a_k, t_k + a_k)$ as a measure of the improbability of y_k being pegged in any subproblem of $\mathcal{P}$. If we want to reduce the size of the branch and bound tree by pegging such variables, then we can consider a branching function that returns the open variable y_j having the largest ϕ_j value. Again, in order to save execution time, we consider the following branching

function that generates the ordering of indices once for the initial solution and uses it for all subproblems.

Branching Function 3 *Define the fitness ϕ_j of the variable y_j as*

$$\phi_j = \begin{cases} \max\{-a_j, t_j + a_j\} & \text{if } y_j \text{ is open in the initial solution,} \\ -\infty & \text{otherwise.} \end{cases}$$

Return the open variable y_j for which ϕ_j is maximal.

In the remainder of this section we will assume the existence of a procedure $FindBranchingVariable$ that takes a partial solution as input, and returns the best variable to branch on.

The pseudocodes for recursive implementations of BnB and BnP algorithms are presented in Figure 3.1. We implemented these algorithms to evaluate their performance on randomly generated problem instances as well as on benchmark problem instances. The BnB algorithm was implemented using Branching Function 1. The BnP algorithms were implemented using each of the three branching functions. Notice that we use preprocessing (using the $PegPartialSolution$ function) for both BnB and BnP algorithms. The pseudocode for the bound used in all the implementations is presented in Figure 3.2. It is an adaptation of a similar bound for general supermodular functions (refer Goldengorin *et al.* [18]) for the SPLP. The algorithms were implemented to allow a maximum execution time of 600 CPU seconds per SPLP instance. The codes were written in C, and run on a Pentium 200 MHz computer running Redhat Linux. The random problem instances were generated in sets of 10 instances each. A problem set is identified by three parameters — the cardinality of the set I (i.e. m), that of the set J (i.e. n), and the density index γ. γ indicates the probability with which an element in the cost matrix has a finite value. Care is taken that, while generating the instances, value each client can be supplied from a plant in at least one of the candidate sites at finite cost regardless of the γ. In each of the randomly generated instances, the fixed costs were chosen from a uniform distribution supported on [10.0, 1000.0], and the finite transportation costs were chosen from a uniform distribution supported on [1.0, 100.0]. The benchmark instances were obtained from the OR-Library [3]. There are twelve SPLP problem instances in this library, four with $m = 16$ and $n = 50$, four with $m = 25$ and $n = 50$, and four with $m = n = 50$. The density index of the transportation cost matrices for all these instances was $\gamma = 1.0$.

```
BnB: main program
begin
    set y ← (#, #, #, . . . , #);
    /* begin Preprocessing */
    y ← PegPartialSolution(y);
    /* end Preprocessing */
    set best ← (1, 1, . . . , 1);
    BnB(y);
    output best;
end.

Procedure BnB (y)
begin
    if y is a complete solution
    begin
        if (z(y) < z(best))
            best ← y;
        return;
    end;
    y_k ← an arbitrary open variable;
    set y_k ← 0;
    if B(y) < z(best)
        BnB(y);
    set y_k ← 1;
    if B(y) < z(best)
        BnB(y);
    return;
end;
```

```
BnP: main program
begin
    set y ← (#, #, #, . . . , #);
    /* begin Preprocessing */
    y ← PegPartialSolution(y);
    /* end Preprocessing */
    set best ← (1, 1, . . . , 1);
    BnP(y);
    output best;
end.

Procedure BnP (y)
begin
    if y is a complete solution
    begin
        if (z(y) < z(best))
            best ← y;
        return;
    end;
    y ← PegPartialSolution(y);
    y_k ← FindBranchingVariable(y);
    set y_k ← 0;
    if B(y) < z(best)
        BnP(y);
    set y_k ← 1;
    if B(y) < z(best)
        BnP(y);
    return;
end;
```

Note:

$best$: the best solution found so far;

$z(\cdot)$: a function to compute the cost of a solution.

$B(\cdot)$: a function to compute the bound from a partial solution.

Figure 3.1: Pseudocodes for BnB and BnP algorithms

Procedure B (**y**: Partial Solution)

begin

$S = \{j : y_j = 0\}$;

$T = \{j : y_j \neq 1\}$;

$l_1 = z(S) - \sum_{k \in T \setminus S} \max\{0, z(S) - z(S + k)\}$;

$l_2 = z(T) - \sum_{k \in T \setminus S} \max\{0, z(T) - z(T - k)\}$;

return $\max\{l_1, l_2\}$;

end;

Note: $z(P)$ is assumed to compute the cost of a solution **y** such that $y_k = 0 \iff k \in P$.

Figure 3.2: Pseudocode for the bound used in the implementations

Tables 3.1 to 3.4 present the results of our computations. Table 3.1 shows the number of problem instances in each data set that were solved by the various algorithms within the stipulated time. Tables 3.2 and 3.3 make a comparative study of the average number of subproblems generated by each of the algorithms and the average execution times, based on the instances in the set that were solved by all the algorithms within the stipulated time. Table 3.4 summarizes our computational experience with the benchmark instances in the OR-Library, presenting both the number of subproblems generated and the execution times required by the algorithms.

Table 3.1: Number of instances in each set solved within 600 CPU seconds

m	n	γ	BnB	BnP Branching Function		
				1	2	3
30	50	0.25	10	10	10	10
		0.50	10	10	10	10
		0.75	10	10	10	10
		1.00	10	10	10	10
40	50	0.25	6	6	6	10
		0.50	10	10	10	10
		0.75	10	10	10	10
		1.00	10	10	10	10
50	50	0.25	1	2	2	8
		0.50	4	7	7	10
		0.75	10	10	10	10
		1.00	10	10	10	10

Table 3.2: The average number of subproblems generated by the algorithms

m	n	γ	Number of common instances	BnB	BnP Branching Function		
					1	2	3
30	50	0.25	10	24330.4	13700.4	13463.4	5573.0
		0.50	10	12769.6	6859.0	6859.0	4448.4
		0.75	10	5426.7	2014.8	1969.6	2635.9
		1.00	10	3301.5	326.7	211.1	203.5
40	50	0.25	6	104624.8	59593.7	53771.3	12887.5
		0.50	10	51927.5	26103.9	26103.9	12218.2
		0.75	10	15420.8	5829.7	5829.7	6400.5
		1.00	10	8799.5	806.6	481.8	498.3
50	50	0.25	*				
		0.50	4	62991.25	29188.25	29188.25	17732.25
		0.75	10	37898.2	14327.6	14327.6	13043.5
		1.00	10	19266.2	1391.9	766.5	932.0

* There was only one instance in common.

Table 3.3: The average execution times required by the algorithms

m	n	γ	Number of common instances	BnB	BnP Branching Function		
					1	2	3
30	50	0.25	10	34.190	27.485	26.843	11.074
		0.50	10	20.515	15.252	15.145	9.252
		0.75	10	8.194	4.966	4.843	5.371
		1.00	10	4.355	0.916	0.698	0.742
40	50	0.25	6	240.357	186.698	164.843	41.708
		0.50	10	131.790	58.297	90.792	40.553
		0.75	10	38.862	22.656	22.407	21.126
		1.00	10	19.482	3.184	2.189	2.535
50	50	0.25	*				
		0.50	4	225.255	152.065	150.498	92.313
		0.75	10	139.090	76.566	75.898	62.987
		1.00	10	62.634	7.626	4.781	6.417

* There was only one instance in common.

Table 3.4: Computational experience with the instances in the OR-Library

Instance	m	n	Number of Subproblems				Execution Times			
			BnB	BnP			BnB	BnP		
				Branching Function				Branching Function		
				1	2	3		1	2	3
cap71	30	50	24	18	19	14	<0.01	<0.01	0.01	0.01
cap72	30	50	37	18	21	13	<0.01	<0.01	0.01	0.01
cap73	30	50	194	130	63	65	0.08	0.07	0.03	0.03
cap74	30	50	63	55	11	37	0.01	0.01	0.01	0.01
cap101	40	50	151	92	92	100	0.05	0.04	0.06	0.07
cap102	40	50	567	325	138	965	0.37	0.31	0.12	0.79
cap103	40	50	2054	589	71	198	1.54	0.62	0.09	0.24
cap104	40	50	943	268	38	72	0.74	0.27	0.09	0.08
cap131	50	50	92543	14148	2167	8016	189.88	35.99	6.24	29.61
cap132	50	50	58564	11234	1226	6992	96.82	22.93	3.29	17.94
cap133	50	50	57697	6459	503	1937	116.88	16.92	1.58	5.92
cap134	50	50	4134	744	125	307	7.57	1.84	0.43	1.05

The tables show that BnP algorithms in general perform much better than BnB algorithms using the same combinatorial bound. They generate less than 60% of the number of subproblems, and require less than 80% of the execution time for instances with sparse transportation cost matrices. For dense transportation cost matrices, the performance of BnP algorithms is much better. They generate less than 10% of the number of subproblems, and require less than 10% of the execution time. The relative performance of these algorithms improve slightly as the size of the instances increase. The BnB algorithm and BnP algorithms using Branching Functions 1 and 2 find instances with low values of γ more difficult to solve. However BnP algorithms using Branching Function 3 solve these instances efficiently. Figure 3.3 presents the improvements by the BnP algorithms over BnB algorithms, both in terms of the number of subproblems generated and in terms of the execution times. The shapes of the component graphs do not change for problem instances of larger size. Based on these observations we can conclude that it is better to run a BnP algorithm that uses Branching Function 2 if the transportation matrix is dense (i.e. $\gamma \gtrapprox 0.6$), and to run a BnP algorithm that uses Branching Function 3 otherwise. This strategy is verified from the results on the instances in the OR-Library. They have dense transportation cost matrices ($\gamma = 1.0$) and BnP algorithms with Branching Function 2 outperform other algorithms for all instances except cap101 (in which BnP with Branching Function 1 outperforms the rest). For problem instances of the same size, all algorithms take more time and generate more subproblems when the optimal solution has a cardinality close to half of

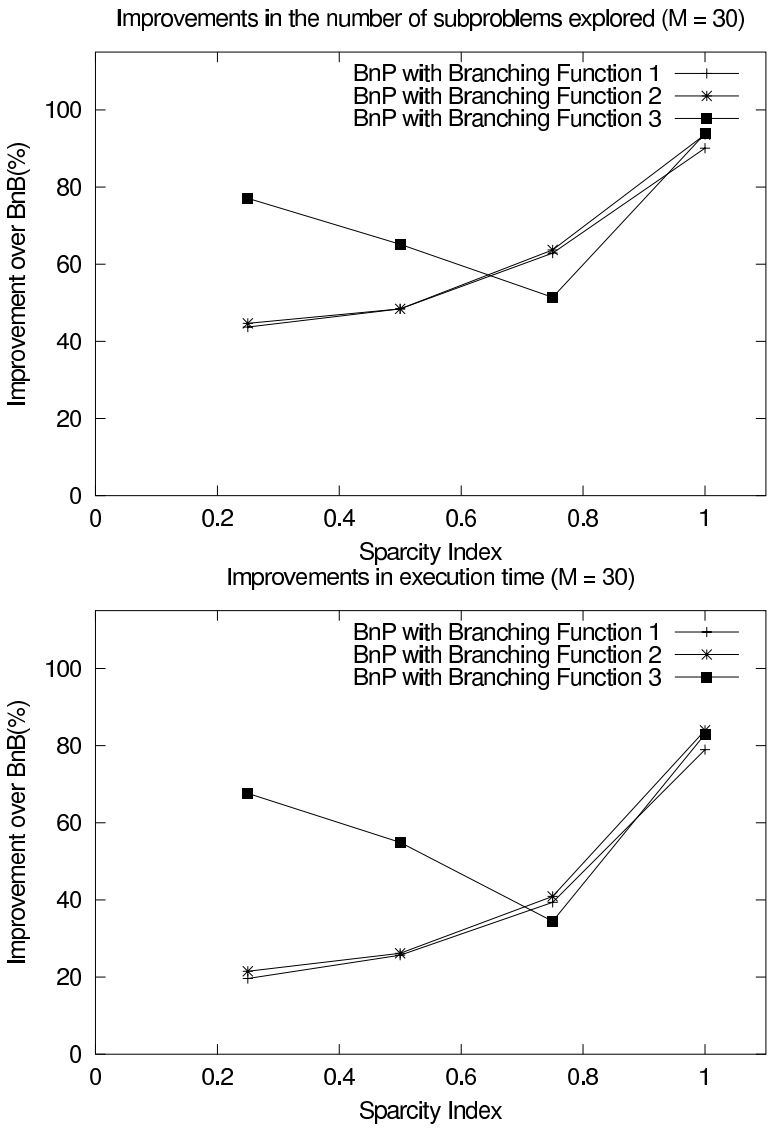

Figure 3.3: Performance of BnP algorithms using BnB algorithm as a basis

the cardinality of I. This can be seen in the problem instances in the OR-Library. The cardinality of the optimal solution to cap101 and cap131 is 15, to cap102 and cap132 is 11, to cap103 and cap133 is 8, and to cap104 and cap134 is 4. Note that cap102 and cap131 are the most difficult to solve among all the instances of the same size for all the algorithms.

4. Summary and Future Research Directions

In this paper we present branch and peg algorithms for the simple plant location problem (SPLP). These algorithms make two improvements on the basic branch and bound scheme. Firstly, for each subproblem generated in the branch and bound tree, a powerful pegging procedure is applied to reduce the size of the subproblem. Secondly, the branching function is based on predictions made using the Beresnev function of the subproblem at hand. We see that branch and peg algorithms comprehensively outperform branch and bound algorithms using the same bound, taking on the average, less than 10% of the execution time of branch and bound algorithms when the transportation cost matrix is dense.

In the first section of the paper we provide a brief introduction to the SPLP, and a brief review on various solution procedures for this problem available in the literature. The second section introduces a pseudo-Boolean polynomial based representation of the problem and the Beresnev function. In Section 3 we develop and test the performance of Branch and Peg algorithms. We demonstrate how the coefficients of the linear terms in the Beresnev function play a crucial role in reducing the size of the current subproblem (Pegging Rule), and allow us to predict the potential aggregation of linear and quadratic terms by pegging a variable. This is used in the design of different branching functions. Our computational experience clearly demonstrates the superiority of branch and peg algorithms over branch and bound algorithms. The main recommendation from the results of the experiment is that branch and peg algorithms should be used to solve SPLP instances. If the transportation cost matrix is sufficiently dense, we recommend a branching function based on a look-ahead scheme, that computes the sizes of the subproblems generated by pegging each variable in the current partial solution, and returns the variable that yields the subproblem of smallest size, as the branching variable (Branching Rule 2). Otherwise, we recommend a branching rule that predicts the variable that is most likely to remain open in all subproblems of the current one, and returns it as a branching variable (Branching Function 3).

The algorithms developed and tested in this paper employ a depth first search scheme. This scheme uses very little computer memory for its execution. However best first search schemes are more useful if we want to generate the minimum number of subproblems. The pegging rule and the branching functions developed in this paper can easily be implemented for branch and bound algorithms using depth first search schemes. It may be interesting to perform computational experiments on branch and peg algorithms using best first search. It may also be interesting to see how the two algorithms compare when other bounds are used.

Table 4.1: Number of subproblems generated in the MBnP algorithm

Size		Density Index (γ)			
m	n	0.25	0.50	0.75	1.0
30	50	4104.5	2863.9	838.1	172.6
40	50	—	8921.6	2185.4	375.2
50	50	—	—	3844.6	631.3

The two new branching functions that we describe in Section 3 need to compute the ordering of the indices only once. This makes the branching functions very time-efficient. But it also makes the implicit assumption that this ordering of indices is also effective for *all* subproblems in terms of the effectiveness of branching. This assumption is not true in general. Consider a modified version of the BnP algorithm (MBnP) that uses Branching Function 2, but in which the ordering of indices is computed for the partial solution in the current subproblem (and not the initial solution). Table 4.1 presents the average number of subproblems generated by MBnP on the same set of randomly generated instances as were used in the previous section. Comparing the entries in this table with the corresponding entries in Table 3.2, we see that the modified algorithm is much more efficient in terms of the number of subproblems generated. However, the time required to compute this branching function is prohibitive, and makes this algorithm useless for all but very small instances. An interesting direction of research is to develop book-keeping techniques that accelerate such branching function computations, so that algorithms like MBnP outperforms the BnP algorithms developed here. It may also be interesting to develop other effective branching functions.

Another interesting direction of research is to incorporate concepts of data correcting (Goldengorin [16], Goldengorin *et al.* [18]) to BnP algorithms. Preliminary computations show that the resulting algorithms are very promising. We plan to experiment with these algorithms in a followup to this work.

References

[1] Balas E, Padberg MW. On the Set Covering Problem. Operations Research 1972;20:1152–1161.

[2] Beasley JE. Lagrangian heuristics for location problems. European Journal of Operational Research 1993;65:383–399.

[3] Beasley JE. OR-Library, `http://mscmga.ms.ic.ac.uk/info.html`

[4] Beresnev VL. On a Problem of Mathematical Standardization Theory. Upravliajemyje Sistemy 1973;11:43–54 (in Russian).

[5] Beresnev VL, Gimadi EKh, Dementyev VT. Extremal Standardization Problems, Novosibirsk, Nauka, 1978 (in Russian).

[6] Christofides N. Graph Theory: An Algorithmic Approach. Academic Press Inc. Ltd., London, 1975.

[7] Cho DC, Johnson EL, Padberg MW, Rao MR. On the Uncapacitated Plant Location Problem. I. Valid Inequalities and Facets. Mathematics of Operations Research 1983;8:579–589.

[8] Cho DC, Padberg MW, Rao MR. On the Uncapacitated Plant Location Problem. II. Facets and Lifting Theorems. Mathematics of Operations Research 1983;8:590–612.

[9] Cornuejols G, Fisher ML, Nemhauser GL. On the Uncapacitated Location Problem. Annals of Discrete Mathematics 1977;1:163–177.

[10] Cornuejols G, Fisher ML, Nemhauser GL. Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms. Management Science 1977;23:789–810.

[11] Cornuejols G, Thizy JM. A Primal Approach to the Simple Plant Location Problem. SIAM Journal on Algebraic and Discrete Methods 1982;3:504–510.

[12] Cornuejols G, Nemhauser GL, and Wolsey LA. The Uncapacitated Facility Location Problem. In: Francis RL, Mirchandani PB (Eds.) Discrete Location Theory. New York:Wiley-Interscience, 1990. p. 119–171.

[13] Dearing PM, Hammer PL, Simeone B. Boolean and Graph Theoretic Formulations of the Simple Plant Location Problem. Transportation Science 1992;26:138–148.

[14] Erlenkotter D. A Dual-Based Procedure for Uncapacitated Facility Location. Operations Research 1978;26:992–1009.

[15] Garfinkel RS, Neebe AW, Rao MR. An algorithm for the M-Median Plant Location Problem. Transportation Science 1974;8:217–236.

[16] Goldengorin B. On the Exact Solution of Standardization Problems by Correcting Algorithms. Soviet Phys. Dokl. 1987;32:432–434.

[17] Goldengorin B. Requirements of Standards: Optimization Models and Algorithms. ROR, Hoogezand, The Netherlands, 1995.
[18] Goldengorin B, Sierksma G, Tijssen GA, Tso M. The Data-Correcting Algorithm for Minimization of Supermodular Functions. Management Science 1999;45:1539–1551.
[19] Goldengorin B, Ghosh D, Sierksma G. Equivalent Instances of the Simple Plant Location Problem. SOM Research Report-00A54, University of Groningen, The Netherlands, 2000.
[20] Guignard M, Spielberg K. Algorithms for Exploiting the Structure of the Simple Plant Location Problem. Annals of Discrete Mathematics 1977;1:247–271.
[21] Hammer PL. Plant Location — A Pseudo-Boolean Approach. Israel Journal of Technology 1968;6:330–332.
[22] Held MP, Wolfe P, Crowder HP. Validation of Subgradient Optimization. Mathematical Programming 1974;6:62–88.
[23] Jones PC, Lowe TJ, Muller G, Xu N, Ye Y, Zydiak JL. Specially Structured Uncapacitated Facility Location Problems. Operations Research 1995;43:661–669.
[24] Krarup J, Pruzan PM. The Simple Plant Location Problem: A Survey and Synthesis. European Journal of Operational Research 1983;12:36–81.
[25] Körkel M. On the Exact Solution of Large-Scale Simple Plant Location Problems. European Journal of Operational Research 1989;39:157–173.
[26] Martin CK, Scharge L. Subset Coefficient Reduction Cuts for 0-1 Mixed Integer Programming. Operations Research 1985;33:505–526.
[27] Morris JG. On the Extent to which Certain Fixed Charge Depot Location Problems can be Solved by LP. Journal of the Operational Research Society 1978;29:71–76.
[28] Mukendi C. Sur l'Implantation d'Èquipement dans un Reseu: Le Problème de m-Centre. Thesis, University of Grenoble, France, 1975.
[29] Pentico DW. The Assortment Problem with Nonlinear Cost Functions. Operations Research 1976;24:1129–1142.
[30] Pentico DW. The Discrete Two-Dimensional Assortment Problem. Operations Research 1988;36:324–332.
[31] Revelle CS, Laporte G. The Plant Location Problem: New Models and Research Prospects. Operations Research 1996;44:864–874.
[32] Schrage L. Implicit Representation of Variable Upper Bounds in Linear Programming. Mathematical Programming Study 1975;4:118–132.
[33] Tripathy A, Süral, Gerchak Y. Multidimensional Assortment Problem with an Application. Networks 1999;33:239–245.

[34] Trubin VA. On a Method of Solution of Integer Programming Problems of a Special Kind. Soviet Math. Dokl. 1969;10:1544–1546.
[35] Van Roy TJ, Wolsey LA. Valid Inequalities for Mixed 0-1 Programs. Discrete Applied Mathematics 1986;14:199–213.
[36] Veselovsky VE. Some Algorithms for Solution of a Large-Scale Allocation Problem. Ekonom. Mat. Metody 1977;12:732–737 (in Russian).